ChatGPT

Deep-Dive Report on Sentence Transformers (SBERT) Documentation

Overview and motivation

Sentence Transformers extend transformer networks (e.g., BERT, RoBERTa) with pooling layers to convert variable-length text into fixed-length embeddings. They support computing semantic similarity, retrieval, clustering, classification, paraphrase mining, topic modelling, translated sentence mining, image search and quantisation. Beyond text, SBERT integrates cross-encoders for re-ranking and sparse encoders (SPLADE) for lexical retrieval, along with training utilities, datasets and losses. Finetuning is essential because different tasks demand different notions of similarity; off-the-shelf embeddings may fail when applied to classification or retrieval [1].

1 Sentence Transformer — Usage

1.1 Computing embeddings

The usage guide begins by showing how to initialise a `SentenceTransformer` model and compute embeddings. You can load any pre-trained model by name or from a local directory; the library automatically selects the best device (CPU or GPU) and offers explicit `device` selection or lists for multi-GPU processing [2]. It supports **prompt templates**: you can define multiple templates and set a default `prompt_name` to modify how inputs are embedded; prompts appear in `config_sentence_transformers.json` when saving the model [2]. The `max_seq_length` attribute controls the maximum input length; you may reduce it (e.g., to save memory) but not extend beyond the base transformer's limit [2].

Batch encoding is efficient: use `model.encode` with `batch_size` and optionally `normalize_embeddings` to produce unit-norm embeddings for dot-product similarity. To leverage multiple GPUs or CPUs for large datasets, pass a list of devices or create a process pool via `start_multi_process_pool` and call `encode_multi_process`. Adjust `chunk_size` and `show_progress_bar` to balance memory and speed【978894366219045†L720-L775】.

1.2 Semantic textual similarity and similarity functions

Similarity can be calculated with several metrics. SBERT's `similarity_fn_name` property accepts `cos_sim`, `dot_sim`, `euclidean` or `manhattan`; these can be set when instantiating the model, via `model.similarity_fn_name = 'dot_sim'`, or by editing the model's config file [3]. Cosine and dot product are equivalent for unit-norm embeddings, but dot product is faster [3]. Methods `model.similarity` and `similarity_pairwise` compute similarity between lists of embeddings or entire matrices; the latter returns a matrix for pairwise comparisons [3].

1.3 Semantic search and retrieval

Bi-encoder semantic search embeds corpus items and queries into the same vector space and then compares them via dot or cosine similarity. For **symmetric search**, encode all texts with the same method. For **asymmetric search**, such as query–document retrieval, call `encode_query` for queries and `encode_document` for documents [4]. A manual search pipeline loads a model, encodes the corpus and queries, computes similarity (e.g., dot product) using `util.semantic_search`, and selects top results [5]. The documentation lists guidelines for speed: normalise embeddings, perform computations on GPU, and use dot product to avoid the expensive norm operation [6]. For large corpora, approximate nearest neighbour libraries (Annoy, FAISS, hnswlib) are recommended; they trade some recall for massive speed gains [6].

Retrieval can be combined with re-ranking. The **retrieve & re-rank** pipeline uses a lexical or bi-encoder retrieval to shortlist candidates and a cross-encoder to re-rank them; this yields high relevance at reasonable cost [7]. Examples in the documentation show simple setups retrieving from Wikipedia and re-ranking with a cross-encoder.

1.4 Clustering and topic modelling

Clustering uses bi-encoder embeddings and unsupervised algorithms:

- **k-Means** clustering requires you to decide the number of clusters; it produces centroids and assigns each embedding to the nearest centroid [8].
- **Agglomerative clustering** starts with each point as its own cluster and merges clusters until a similarity threshold is reached [9].
- **Fast clustering** is a scalable algorithm for millions of sentences; you can set a similarity threshold and a minimal community size [10].
- **Topic modelling** transforms clusters into interpretable topics; SBERT works with libraries such as BERTopic and Top2Vec to discover themes [11].

1.5 Paraphrase mining and translated sentence mining

`paraphrase_mining` compares all sentences in a corpus to identify paraphrase pairs; it uses cosine similarity and returns pairs with scores above a threshold. Parameters like `query_chunk_size`, `corpus_chunk_size`, `top_k` and `score_function` control speed and memory use; duplicates are removed, but note that the most similar sentence to A may not be B [12].

Translated sentence mining finds parallel sentences across languages using margin-based mining. It encodes sentences with a multilingual model (e.g., LaBSE), finds the k nearest neighbours in both directions, scores pairs using a margin function and selects those above a threshold around 1.2–1.3 [13]. Example scripts illustrate mining BUCC 2018 and JW300 datasets.

1.6 Image search and multimodal embeddings

SBERT provides CLIP-based models that encode images and texts into a shared space. To use them, install the `transformers` library and load a CLIP model (e.g., `sentence-transformers/clip-ViT-B-32`). The model's `.encode` accepts either text or images (PIL objects or file paths) and produces embeddings;

you then compute similarities to perform text-to-image or image-to-text search. Fine-tuning on your own domain is possible [14]. The documentation links notebooks demonstrating multilingual CLIP search, image clustering, duplicate detection and zero-shot classification [15].

1.7 Embedding quantisation

To deploy SBERT at scale, you can reduce embedding size via quantisation:

- **Binary quantisation** converts 32-bit floats to 1-bit values, enabling retrieval by Hamming distance. A rescoring step using full-precision embeddings preserves up to 96 % of performance while reducing memory 32× [16].
- **Scalar (int8) quantisation** uses a calibration dataset or range to map float values to 8-bit integers; retrieval performance improves if you rescore top results with original floats [17].
- **Combined quantisation** performs binary retrieval and scalar rescoring for memory savings and high accuracy [18].

1.8 Custom model construction

A SentenceTransformer is essentially a sequence of modules (e.g., `models.Transformer`, `models.Pooling`, `models.Dense`, `models.Normalize`). The custom models page shows how to manually construct the `all-MiniLM-L6-v2` model: load a transformer backbone, attach a pooling layer to compute mean, max, and CLS token embeddings, optionally add a dense layer and normalisation, then create the SentenceTransformer object. Saving the model produces files `modules.json`, `config_sentence_transformers.json` and module-specific configs; `config_sentence_transformers.json` stores metadata such as library version and default prompts [19]. When loading a pure transformer model with SentenceTransformer, the framework automatically adds a pooling module.

1.9 Speeding up inference and evaluation

Although not captured in citations, the usage pages contain guidance for converting models to ONNX or OpenVINO, using FP16 or BF16 to increase throughput, and selecting appropriate batch sizes. For evaluation, the MTEB benchmark is recommended: it comprises tasks spanning retrieval, classification, clustering, re-ranking and summarisation. Use `pip install mteb` and run `evaluation.run(model, dataset_name)`; avoid evaluating on MTEB during training to prevent overfitting [20].

2 Sentence Transformer — Pretrained Models

2.1 Original and semantic search models

SBERT releases many pre-trained models hosted under the Sentence-Transformers organisation on Hugging Face. These are grouped by task:

- **Semantic Search** models such as `multi-qa-mpnet-base-dot-v1` and `msmarco-MiniLM-L6-cos-v5` produce normalised embeddings optimised for dot or cosine similarity; performance metrics from retrieval benchmarks are provided [21].

- **General-purpose** models like `all-MiniLM-L6-v2` provide balanced performance for clustering and classification.

Users should consult the MTEB leaderboard when selecting models but consider trade-offs: large models may be overkill for short queries [21].

2.2 Multilingual and domain-specific models

Multilingual models include `distiluse-base-multilingual-cased-v1`, `distiluse-base-multilingual-cased-v2` and `paraphrase-multilingual-MiniLM-L12-v2`; these support 50+ languages and work well for cross-lingual semantic similarity. LaBSE (Language-agnostic BERT Sentence Embedding) is good for bitext mining but less optimal for general similarity [22]. For scientific text, `allenai/specter` encodes titles and abstracts and performs well in scientific similarity tasks.

2.3 Image and instruction models

Image and text models include CLIP variants; `clip-ViT-B-32-multilingual-v1` extends CLIP to 50+ languages. Top-1 zero-shot ImageNet accuracy is given for each variant. **INSTRUCTOR models** combine textual instructions with inputs to produce context-aware embeddings. Usage requires prepending an instruction (e.g., “Represent the Wikipedia document for retrieval: ”) to every input; some models require setting `Pooling(include_prompt=True)` to include the instruction in the final embedding [23].

3 Sentence Transformer — Training

3.1 Why finetune?

Pretrained models capture generic semantics but cannot adapt to different tasks—classification, similarity, retrieval or ranking—without fine-tuning. The training overview emphasises that finetuning yields substantial performance gains: for example, a news classifier should map technology articles close together while pushing away unrelated news, a requirement not met by generic embeddings [1].

3.2 Training pipeline components

SBERT training comprises:

- **Model** – choose a pretrained model or build one. When building from scratch, create a `models.Transformer` for your domain or language and add pooling. The documentation provides code to load `bert-base-uncased` and attach a mean pooling layer. You can also use static embedding models for speed at the cost of reduced semantic quality [24].

- **Dataset and format** – load datasets via `datasets.load_dataset`. SBERT provides curated datasets with the `sentence-transformers` tag; examples include All NLI, MS MARCO triplets, Quora duplicates, SQuAD and Amazon reviews [25]. The dataset format must match the loss: `label` or `score` columns for similarity losses; anchor–positive pairs for in-batch negatives; triplets for triplet losses. You may reorder or drop extraneous columns using `.select_columns` or `.remove_columns` [26].

- **Loss functions** – The loss table maps data shapes to losses. Use `MultipleNegativesRankingLoss` (MN-RL) or `TripletLoss` for anchor–positive pairs; `CosineSimilarityLoss`, `CoSENTLoss` or `AnglELoss` for scored pairs; `MarginMSELoss` or `MSELoss` for distillation; `MultipleNegativesRankingLossCached` for large datasets; `TripletLoss` for explicit triplets [27]. Loss modifiers such as `MatryoshkaLoss` enforce truncated embedding robustness and `AdaptiveLayerLoss` encourages layered representations [28]. Distillation losses require teacher embeddings or soft labels [29].

- **Training arguments** – Use `SentenceTransformerTrainingArguments` to set hyperparameters. Important parameters include `learning_rate`, scheduler and `warmup_ratio`, `num_train_epochs`, `per_device_train_batch_size`, `fp16`/`bf16` for mixed precision, `batch_sampler` (use `BatchSamplers.NO_DUPLICATES` for in-batch negatives), evaluation and save frequency [30].

- **Evaluator** – Choose evaluators appropriate to your task: binary classification, embedding similarity, information retrieval, MSE, paraphrase mining, re-ranking, translation or triplet evaluation [31]. Multiple evaluators can be combined with `SequentialEvaluator`.

- **Trainer** – `SentenceTransformerTrainer` orchestrates training. Pass the model, training dataset, loss, arguments and optional evaluator. For multi-dataset training, supply a dictionary of datasets (and optionally a dictionary of losses) to sample from each dataset proportionally. After training, call `model.save_pretrained` or `model.push_to_hub` [30].

3.3 Parallel and cross-lingual training

SBERT supplies dedicated examples for many tasks: STS (Semantic Textual Similarity), natural language inference, paraphrase detection, Quora duplicates, MS MARCO retrieval, Matryoshka embeddings, adaptive layers, multilingual models and model distillation. The multilingual training example uses a teacher–student distillation framework to extend an English model to new languages. It takes parallel sentences (source and translation) and minimises the MSE between the student's embeddings and the teacher's embeddings for both sentences; evaluation uses `MSEEvaluator` and `TranslationEvaluator`. Data can be loaded from the `sentence-transformers/parallel-sentences` collections [32] [33].

4 Dataset Overview

Beyond the general dataset guidance, SBERT lists many pre-existing datasets on the Hugging Face Hub tagged for sentence transformers. They include GooAQ (question–answer pairs), Yahoo Answers, MS MARCO triplets, Stack Exchange duplicates, SQuAD, WikiHow, Amazon Reviews, Natural Questions, Amazon QA and scientific corpora (S2ORC) [25]. The dataset overview shows how to load a dataset using `load_dataset("qid_md10_1_triplets")` or similar, inspect the columns and remove extraneous columns with `.remove_columns` [34].

5 Loss Overview

SBERT's loss overview matches input formats to loss functions [27] and provides guidance for custom losses [35]. Notable losses include:

- **MultipleNegativesRankingLoss (MN-RL)** – A contrastive loss that pushes positive pairs closer and in-batch negatives away; widely used for retrieval and unsupervised clustering. The cached variant allows training on large corpora using a buffer to provide more negatives.
- **TripletLoss** – Uses explicit anchor–positive–negative triplets; best when high-quality hard negatives are available.
- **CosineSimilarityLoss/CoSENTLoss/AnglELoss** – For pairwise data with similarity scores; CoSENT and AnglE improve ranking by promoting larger margins between correctly and incorrectly ordered pairs [36].
- **MarginMSELoss** – Distillation loss for teacher–student training; uses margin between positive and negative teacher similarities.
- **DistillKLDivLoss** – Distils soft labels (probability distributions) from a teacher cross-encoder [29].
- **MatryoshkaLoss and AdaptiveLayerLoss** – Loss modifiers for producing embeddings robust to truncation or using layered models [28].

6 Training Examples

The training examples repository provides complete scripts for numerous tasks. Key themes include:

- **STS/NLI/Paraphrase** – Fine-tune on SNLI, MNLI and STSbenchmark datasets for semantic similarity; some scripts perform unsupervised training using SimCSE or contrastive losses.
- **Quora duplicates** – Train classification or retrieval models to identify duplicate questions.
- **MS MARCO retrieval** – Example training on the MSMARCO passage ranking dataset with `MultipleNegativesRankingLoss` to build retrieval models; includes demonstration of distributed training.
- **Matryoshka embeddings** – Train models to produce truncated embeddings that retain semantic quality across various sizes.
- **Adaptive layers** – Train models that dynamically select a subset of transformer layers based on input length, improving efficiency.
- **Multilingual extension** – Distil an English model into a multilingual model using parallel sentences [32].
- **Distillation** – Transfer knowledge from cross-encoders to bi-encoders using margin-MSE or KL losses; used to create retrieval models with fewer parameters.

7 Cross Encoder

7.1 Usage and comparison with bi-encoders

Cross-encoders take a pair of sequences as input and output a similarity or relevance score via a classification head. They do not produce standalone embeddings; they jointly process the sequences and therefore capture richer interactions but are slower. The documentation emphasises that cross-encoders are best used to re-rank a shortlist of candidates retrieved by a bi-encoder [37]. To use a cross-encoder for inference, load a model (e.g., `cross-encoder/ms-marco-MiniLM-L6-v2`) and call

`model.predict([(query, doc1), (query, doc2), ...])`; if using a classification head with two labels, take `sigmoid` of the logits to map scores to [0,1] [38]. For retrieval pipelines, embed all documents using a bi-encoder, retrieve top candidates, then re-rank them using the cross-encoder [39].

7.2 Pretrained cross-encoder models

Pre-trained cross-encoders include:

- **MS MARCO rerankers** – Models of various sizes (TinyBERT, MiniLM, BERT-base, MS-Dial) trained on passage ranking; performance metrics (NDCG@10, MRR@10, docs/sec) are provided [40].
- **STS and Quora duplicates models** – Cross-encoders trained on STSbenchmark and Quora duplicate question pairs; output similarity scores or duplicate probabilities [40].
- **NLI models** – Cross-encoders fine-tuned on MNLI; predictions across classes entailment, contradiction and neutral [40]. Example code shows mapping cross-encoder scores to labels.

Community models (e.g., from BAAI, JinaAI) are available but not individually summarised; the page lists them as further options [41].

7.3 Training cross-encoders

The training pipeline for cross-encoders mirrors sentence transformer training: choose a base transformer (e.g., BERT-base), add a classification head if missing; load or build a dataset with pairs and labels; select a loss (e.g., cross-entropy or KL divergence); configure `CrossEncoderTrainingArguments`; choose an evaluator; and run `CrossEncoderTrainer`【550292593396418†L538-L736】. Finetuning cross-encoders is important to avoid performance degradation, especially when transferring from classification to ranking tasks. The dataset format guidelines emphasise matching the number of model output labels to the number of label classes and removing unused columns. Hard negative mining is critical: use `util.mine_hard_negatives` to find challenging negative examples using a retrieval model or cross-encoder; the function allows controlling margins and negative types and can employ FAISS for efficiency [42].

7.4 Loss overview and examples

The cross-encoder loss overview is similar to the sentence transformer one but focused on classification losses (e.g., cross-entropy, pairwise ranking losses). The examples include scripts for STS, NLI, Quora duplicates and MS MARCO rerankers. A reranking-specific page explains how to combine cross-encoder scores with retrieval scores and emphasises using a Sigmoid or softmax to normalise logits.

8 Sparse Encoder (SPLADE)

8.1 Usage and sparse embeddings

Sparse encoders produce vectors the size of the vocabulary, with most values zero. They implement lexical matching using neural models. The usage page shows how to load a `SparseEncoder` model (e.g., `coastalcph/mt5-splade-v3`), call `.encode` on texts to produce sparse representations and compute similarities via `.similarity` (dot or cosine). SBERT supports multi-process or multi-GPU encoding by

passing a list of devices or launching a process pool; `chunk_size` helps control memory [43]. You can compute pairwise similarities or call `similarity_pairwise` similar to dense models [44].

8.2 Controlling sparsity and interpretability

SPLADE models use activation functions (e.g., ReLU and log exp) to produce weights for each token. You can limit the number of active dimensions using the `max_active_dims` parameter: this truncates the highest-weighted dimensions and dramatically reduces memory usage at a modest accuracy cost. For example, limiting active dimensions to 32 saved about 43 % of memory in the example code [45]. The `SparseEncoder.sparsity_stats` method reports the average number of active dimensions and non-zero ratio.

SPLADE models are interpretable: you can call `model.decode(sparse_embedding, top_k)` to obtain the top contributing tokens and their weights. This helps understand which terms drive similarity; the documentation shows decoding the top three tokens for each of several example sentences [46].

8.3 Semantic similarity and search with sparse models

For sparse embeddings, semantic textual similarity is computed using the same metrics (`dot`, `cosine`, `euclidean`, `manhattan`). Example code encodes two lists of sentences and computes similarity matrices, then selects the most similar pairs [47]. You can change the similarity metric by setting `model.similarity_fn_name` or passing it to functions [44].

Manual semantic search follows similar steps to dense retrieval: encode the corpus and queries, compute similarities, sort results and optionally inspect tokens. The manual search example uses `util.semantic_search` with sparse embeddings and prints both relevance scores and the influential tokens per query [48].

8.4 Pretrained sparse models

SBERT hosts several SPLADE models: **core SPLADE** models (e.g., `naver/splade-cocondenser`), **inference-free SPLADE** that skip certain inference layers, and community models. These are grouped by training dataset (e.g., MSMARCO, Naver), language (English or multilingual), and architecture (BERT, MT5). Each model card lists vocabulary size, active dimensions, and retrieval metrics. Users should choose based on domain, language and resource constraints.

8.5 Training sparse encoders

Training a SPLADE model requires modifications to the standard pipeline:

- **Model** – use a transformer backbone with `SparseLayer` or `SpladePooling` to generate sparse scores.
- **Dataset and format** – as for dense models, but you should align the tokeniser and vocabulary with the teacher model if performing distillation. Inputs typically consist of (query, document) pairs.
- **Loss functions** – SPLADE uses variants of `MultipleNegativesRankingLoss` or `SparseMultipleNegativesRankingLoss` that encourage lexical sparsity. Distillation losses, such

as `KDTripletLoss` or `LexicalDistillationLoss`, transfer knowledge from dense teachers. The `CSRLoss` (Contrastive Softmax Ranking Loss) is also used for information retrieval.
- **Training arguments** – same as dense models but emphasise controlling sparsity with `max_active_dims` and monitoring memory usage.
- **Evaluators** – the `SparseInformationRetrievalEvaluator` computes retrieval metrics (accuracy, precision, recall, MRR, NDCG) and can print sparsity statistics [49].

Example training scripts include distillation of BERT-based SPLADE models on MS MARCO using sparse MNRL, and domain adaptation of SPLADE to multi-lingual corpora. Hard negative mining and multi-dataset training are applicable here too.

9 Package Reference

SBERT's package reference documents classes and functions:

- **SentenceTransformer** – `__init__` to load or build models; `.encode` for embeddings; `.similarity` / `.similarity_pairwise`; `.fit` for legacy training; `.save_pretrained` and `.push_to_hub`; `start_multi_process_pool`; `.encode_multi_process`; `.similarity_fn_name` property; and utilities for quantisation.
- **CrossEncoder** – accepts text pairs; `.predict` outputs scores or class labels; `.save_pretrained`; training arguments and training methods. Importantly, cross-encoders do not provide embeddings and must not be used for vector search [37].
- **SparseEncoder** – `.encode`, `.similarity`, `.decode`, `.sparsity_stats`; training modules like `SpladePooling` and losses such as `CSRLoss`.
- **util** – helper functions for semantic search, paraphrase mining, similarity scoring, negative mining and prompt generation. `util.semantic_search` performs dense or sparse search; `util.paraphrase_mining` implements paraphrase mining; `util.mine_hard_negatives` mines negative examples using retrieval models and cross-encoders [42].

10 Practical guidance for training and extension

To train high-quality embeddings for a new task:

1. **Select a base model** appropriate to your language and domain (e.g., `mpnet-base` for English, `paraphrase-multilingual-MiniLM-L12-v2` for multilingual tasks). Use the MTEB leaderboard for comparison.

2. **Prepare your dataset** to match the chosen loss. For general retrieval, create anchor–positive pairs or triplets. For scoring tasks, prepare pairs with similarity scores. For cross-lingual extension, assemble parallel sentences (source and translation). Remove extraneous columns and ensure correct ordering [26].

3. **Choose a loss** suited to your supervision: MN-RL for unlabeled pairs; CosineSimilarityLoss or CoSENTLoss for scored pairs; TripletLoss for triplets; Distillation losses for teacher–student transfer

[27]. For cross-lingual training, use MSELoss or MarginMSELoss to match teacher and student embeddings for both languages [32].

4. **Set training arguments** with sensible defaults: learning rate (2e-5), warmup ratio (0.1), epochs (1–3), batch size (16–32), `fp16` or `bf16` for mixed precision, `batch_sampler="no_duplicates"` to avoid duplicates in in-batch negatives [30].

5. **Add evaluators** relevant to your task. Use `InformationRetrievalEvaluator` for retrieval, `EmbeddingSimilarityEvaluator` for similarity, `TripletEvaluator` for triplet tasks, `MSEEvaluator` and `TranslationEvaluator` for multilingual distillation [31].

6. **Train** using `SentenceTransformerTrainer` or `CrossEncoderTrainer`. For multi-dataset training, supply datasets as a dictionary to sample from each dataset. Save and share models using `model.save_pretrained()` or `model.push_to_hub()` [30].

7. **Evaluate** after training. For dense models, run MTEB or custom evaluation tasks to avoid overfitting during training [20]. For sparse models, use `SparseInformationRetrievalEvaluator` [49].

8. **Quantise** if necessary to deploy at scale; combine binary retrieval with scalar rescoring for efficient memory use [18].

9. **Extend to new languages** via teacher–student distillation: use parallel sentences, compute teacher embeddings with an English model, and train a multilingual student using MSE or MarginMSE loss [32]. Evaluate cross-lingual models on translation tasks and STS datasets [50].

Conclusion

SBERT provides a comprehensive ecosystem for constructing, training and deploying sentence embeddings. The modular architecture supports dense, sparse and cross-encoder models, each suited to different trade-offs between speed and accuracy. Fine-tuning is fundamental: generic embeddings seldom align with task-specific notions of similarity [1]. With the curated datasets, loss functions, training scripts and evaluation tools documented here, practitioners can build state-of-the-art models for semantic search, clustering, classification, paraphrase detection, question answering and cross-lingual applications. The framework's extensibility—especially the ability to distil new languages and reduce model size via quantisation—makes SBERT a versatile toolkit for research and production.

---

1 24 26 30 31 Training Overview — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/training_overview.html

2 Computing Embeddings — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/computing-embeddings/README.html

3 Semantic Textual Similarity — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/usage/semantic_textual_similarity.html

4 5 6 Semantic Search — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/semantic-search/README.html

7 Retrieve & Re-Rank — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/retrieve_rerank/README.html

8 9 10 11 Clustering — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/clustering/README.html

12 Paraphrase Mining — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/paraphrase-mining/README.html

13 Translated Sentence Mining — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/parallel-sentence-mining/README.html

14 15 Image Search — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/image-search/README.html

16 17 18 Embedding Quantization — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/embedding-quantization/README.html

19 Creating Custom Models — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/usage/custom_models.html

20 Evaluation with MTEB — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/usage/mteb_evaluation.html

21 22 23 Pretrained Models — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/pretrained_models.html

25 34 Dataset Overview — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/dataset_overview.html

27 28 29 35 36 Loss Overview — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/loss_overview.html

32 33 50 Multilingual Models — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/training/multilingual/README.html

37 38 39 Cross-Encoders — Sentence Transformers documentation
https://sbert.net/examples/cross_encoder/applications/README.html

40 41 Pretrained Models — Sentence Transformers documentation
https://sbert.net/docs/cross_encoder/pretrained_models.html

42 Training Overview — Sentence Transformers documentation
https://sbert.net/docs/cross_encoder/training_overview.html

43 45 46 Computing Sparse Embeddings — Sentence Transformers documentation
https://sbert.net/examples/sparse_encoder/applications/computing_embeddings/README.html

44 47 Semantic Textual Similarity — Sentence Transformers documentation
https://sbert.net/examples/sparse_encoder/applications/semantic_textual_similarity/README.html

48 Semantic Search — Sentence Transformers documentation
https://sbert.net/examples/sparse_encoder/applications/semantic_search/README.html

49 Sparse Encoder Evaluation — Sentence Transformers documentation
https://sbert.net/examples/sparse_encoder/evaluation/README.html